You've implemented a Spring Boot Kafka producer. However, messages are not being delivered to the Kafka topic. What could be the potential reasons?

Several issues could cause the failure of Kafka message delivery:

Kafka broker connectivity issues: Ensure that the Kafka producer is able to connect to the Kafka brokers (check bootstrap-servers configuration).

Incorrect topic name: Double-check that the topic name in the producer configuration matches the actual topic in Kafka.

Serializer issues: Ensure that the correct serializers (e.g., StringSerializer, IntegerSerializer) are configured for the producer.

Kafka producer configuration: Verify the producer's configurations like acks, retries, key.serializer, and value.serializer are set correctly.

Topic existence: Ensure that the Kafka topic exists (if auto-creation is not enabled).

Network/firewall issues: Check for network/firewall issues between the Spring Boot application and Kafka brokers.

Troubleshooting Steps:

Enable debug logging in Spring Boot to check the Kafka producer logs.

Use kafka-console-consumer to verify whether the messages are being published to the topic.

==========

You are building a Spring Boot application that consumes messages from a Kafka topic. After a message is consumed, the consumer application should perform some actions like saving data into a database. However, the data is not being saved. How would you troubleshoot this?

To troubleshoot the issue, you could follow these steps:

Consumer Configuration: Check the configuration of the consumer (e.g., group.id, auto.offset.reset, enable.auto.commit) to ensure it's correctly set.

Kafka Consumer Logs: Enable logging to see if messages are being consumed and whether there are any errors during consumption.

Message Processing Logic: Verify that the message processing logic, including database operations, is correctly implemented. This could involve checking the database connection, SQL statements, and transaction management.

Database Connectivity: Ensure that the application is connected to the correct database and that there are no issues like network failures or invalid credentials.

Error Handling: Check if there are any exceptions thrown during the message processing, especially during database operations. Look into @Retryable or other retry mechanisms.

Spring Kafka Listener Annotation: Make sure the method is correctly annotated with @KafkaListener and the listener is properly registered in the Spring Boot context.

==========

You need to consume messages from a Kafka topic with a delay. For example, after consuming the message, you want to wait for a few seconds before processing it. How would you implement this in Spring Boot Kafka?

Introduce a delay in your Kafka consumer by using Thread.sleep() or by scheduling the processin logic with an additional delay.

```
@KafkaListener(topics = "your-topic")
public void consumeMessage(String message) throws InterruptedException {
   // Delay for 5 seconds
   Thread.sleep(5000);
   // Process the message after the delay
   processMessage(message);
}
```

However, be cautious when using Thread.sleep() as it can block the thread and reduce the performance of the consumer. A more scalable approach might be to use a separate thread or a task scheduler to handle the delayed processing without blocking the consumer.

You are processing messages from Kafka in a Spring Boot application. The consumer is running in a clustered environment with multiple instances. How does Kafka ensure that each message is processed exactly once, and what configuration changes would you make in your Spring Boot application?

Kafka provides at-least-once delivery semantics by default, meaning messages are processed at least once, but they could be processed more than once in certain

failure scenarios (e.g., consumer crashes before committing an

offset). To ensure exactly-once processing, you need to configure both

Kafka and Spring Boot appropriately:

Steps to implement exactly-once processing:

Kafka Configuration: Enable idempotence on the Kafka producer to avoid producing duplicates:

spring.kafka.producer.properties.enable.idempotence=true

Enable transactional processing on the Kafka producer for atomicity:

spring.kafka.producer.transaction-id-prefix=tx-

Spring Kafka Consumer Configuration:Use manual offset management and commit offsets only after the message

has been successfully processed:

spring.kafka.consumer.enable-auto-commit=false

Use Kafka transactions on the consumer side to ensure that if processing fails, no offsets are committed until the message has been fully processed.

Use @KafkaListener with the Acknowledgment parameter to commit offsets manually:

```
@KafkaListener(topics = "your-topic")
public void consumeMessage(String message, Acknowledgment
acknowledgment) {
   // Process the message
   processMessage(message);

   // Commit the offset manually
   acknowledgment.acknowledge();
}
```

Kafka Transactional Consumer:
Use Kafka transactions on the consumer side to ensure that if processing fails, no offsets are committed until the message has been fully processed.

You are required to implement a Spring Boot Kafka producer that needs to send messages to multiple topics based on some dynamic criteria. How would you implement this?

You can implement dynamic topic selection by programmatically determining the topic name before sending the message. Here's how yo can achieve it:

Inject KafkaTemplate into your service.

Use send method on KafkaTemplate with the topic name dynamically chosen based on the message content.

```
@Service
public class KafkaProducerService {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;
    public void sendMessage(String message) {
        String topic = determineTopicBasedOnMessage(message); // Determine topic dynamically
        kafkaTemplate.send(topic, message);
    }
    private String determineTopicBasedOnMessage(String message) {
        // Logic to choose the topic based on message content
        return message.contains("urgent") ? "urgent-topic" : "regular-topic";
    }
}
```

This approach ensures that the message can be routed to different topics based on dynamic conditions like message type or content.

Your Spring Boot application has a Kafka consumer with a high volume of messages. The consumer is falling behind in processing the messages, and you need to scale the consumer application to handle the load. What would you do?

To scale your Kafka consumer application and handle the high message volume, you can:

Increase consumer instances: Kafka allows multiple consumers in the same consumer group to process differen partitions in parallel. You can increase the number of instances of your Spring Boot application to scale horizontall

Partitioning: Ensure that the Kafka topic is partitioned, as each partition is consumed by only one consumer in a group at a time. By increasing the number of partitions, you can parallelize the consumption.

Concurrency in @KafkaListener: You can specify multiple threads to consume messages concurrently from a single consumer instance using @EnableAsync or @KafkaListener with the concurrency property.

```
@KafkaListener(topics = "your-topic", concurrency = "3")
public void consumeMessages(String message) {
    processMessage(message);
}
```

Message batching: Consider batching messages to process them in bulk rather than individually for better throughput.

In your Spring Boot Kafka application, you need to ensure that the messages are not lost in case of a Kafka broker failure. How would you ensure message durability?

To ensure message durability, you should configure the producer and consumer with the following best practices:

Producer Side (Durability): Set acks=all to ensure that the producer waits for acknowledgment from all replicas of a Kafka partition before confirming message receipt. This ensures the message is written to multiple replicas and is durable.

spring.kafka.producer.acks=all

Enable retries and idempotence on the producer to handle transient failures:spring.kafka.producer.retries=3

spring.kafka.producer.properties.enable.idempotence=true

Consumer Side: Use manual offset commits to ensure that the consumer can reprocess messages from where it left off in case of failure.

spring.kafka.consumer.enable-auto-commit=false

Your Spring Boot application processes sensitive data from a Kafka topic. How would you implement encryption and decryption for messages in Kafka?

For processing sensitive data, you should ensure that the messages are encrypted before they are sent to Kafka and decrypted once they are consumed. Here's how you can implement encryption and decryption in Spring Boot Kafka:

Encrypt the message before sending: Implement encryption in your Kafka producer before sending the message to Kafka. You can use libraries like JCE (Java Cryptography Extension) to encrypt the message.

Decrypt the message after consuming: Similarly, decrypt the message when it is consumed by the Kafka listener.

Example of a producer with encryption:

```
@Service
public class KafkaProducerService {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;
    public void sendMessage(String message) {
        String encryptedMessage = encryptMessage(message);
        kafkaTemplate.send("your-topic", encryptedMessage);
    }

    private String encryptMessage(String message) {
        // Implement encryption logic here
        return Base64.getEncoder().encodeToString(message.getBytes());
    }}
```

Your Spring Boot application processes sensitive data from a Kafka topic. How would you implement encryption and decryption for messages in Kafka?.....

Example of a consumer with decryption:

```
@KafkaListener(topics = "your-topic")
public void consumeMessage(String encryptedMessage) {
   String decryptedMessage = decryptMessage(encryptedMessage);
   // Process the decrypted message
}

private String decryptMessage(String encryptedMessage) {
   // Implement decryption logic here
   return new String(Base64.getDecoder().decode(encryptedMessage));
}
```

Note: Depending on your requirements, you may also want to consider SSL encryption for Kafka communication between producers, brokers, and consumers.

Your Spring Boot Kafka consumer has high throughput, and the consumer group is processing messages slowly due to inefficient processing logic. How would you optimize the consumer's performance while maintaining message integrity?

Optimizing the Kafka consumer for high throughput while ensuring message integrity requires tuning both the Kafka consumer settings and your message processing logic.

Here are a few approaches to optimize the consumer's performance:

Increase the number of partitions: More partitions allow Kafka to distribute the message load more evenly across consumers in the consumer group, improving parallelism.

Concurrency in listeners: Use Spring Kafka's @KafkaListener with concurrency to enable multi-threading and parallel message processing within a single consumer instance.

```
@KafkaListener(topics = "your-topic", concurrency = "5")
public void consumeMessages(String message) {
   processMessage(message);
}
```

Your Spring Boot Kafka consumer has high throughput, and the consumer group is processing messages slowly due to inefficient processing logic. How would you optimize the consumer's performance while maintaining message integrity?.....

Efficient message processing: Make sure your message processing logic is efficient. For instance, avoid making multiple database calls for each message. Batch your processing or use asynchronous processing if necessary.

Offload to background processing: For long-running tasks, consider offloading them to background jobs using @Async or Spring's TaskExecutor. This will allow the Kafka consumer thread to focus on consuming more messages without waiting for background tasks to finish.

Tune consumer configurations:

Fetch size: Increase fetch.min.bytes and fetch.max.wait.ms to pull more messages per request.

Max poll interval: Increase max.poll.interval.ms to allow more time for processing before the consumer is considered slow.

Batching: Configure batching on the consumer side (max.poll.records) to process multiple records in one poll.

```
spring.kafka.consumer.fetch-min-bytes=10000
spring.kafka.consumer.fetch-max-wait=500
spring.kafka.consumer.max-poll-records=500
```

You are building a Spring Boot Kafka application that must ensure message ordering across multiple partitions. How would you design th system?

Kafka guarantees message order only within a single partition. If you have multiple partitions for your Kafka topic, the messages may be processed out of order across different partitions.

To maintain ordering, you can use the following strategies:

Single partition per key: If maintaining order for specific types of messages (e.g., for a customer ID or order ID) is required, ensure that all messages related to the same key are sent to the same partition. This can be done using a custom partitioner or by ensuring the producer uses a consistent key.

Example of sending a message with a key:

```
@Service
public class KafkaProducerService {

   @Autowired
   private KafkaTemplate<String, String> kafkaTemplate;

   public void sendMessage(String customerId, String message) {
      kafkaTemplate.send("your-topic", customerId, message); // Customer ID as the key to ensure order within that partition
   }
}
```

You are building a Spring Boot Kafka application that must ensure message ordering across multiple partitions. How would you design this system?...

Custom Partitioner: Implement a custom Kafka producer partitioner to control the partition assignment logic based on the message key.

Example:

```
public class CustomPartitioner implements Partitioner {

  @Override
  public void configure(Map<String, ?> configs) {}

  @Override
  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
    // Custom partition logic based on key (e.g., hash customer ID)
    return Math.abs(key.hashCode()) % cluster.partitionCountForTopic(topic);
  }

  @Override
  public void close() {}
}
```

Reorder messages at the consumer side: In some use cases where the message order is important, and if they are processed out of order, you may need to introduce an additional ordering mechanism in your consumer logic. For instance, each consumer can keep track of the last processed message and wait for messages to arrive in order.

🔄 Producer Side

1. **Scenario:** *A Kafka producer sends messages, but the broker is down.*

Q: What happens, and how do you handle it?

A:

- Kafka producer will retry sending messages (based on `retries` config).
- If retries are exhausted, an exception is thrown.
- You can set `acks=all`, `retries > 0`, and enable idempotence (`enable.idempotence=true`) to make sending more reliable.
- Also consider exponential backoff and async error handling.

---

2. **Scenario:** *Producer is sending duplicate messages.*

Q: How do you prevent duplicates?

A:

- Enable **idempotent producer** by setting `enable.idempotence=true`.
- Kafka ensures exactly-once delivery per topic-partition.
- Combine this with `acks=all` and appropriate retries.

3. **Scenario:** *You want to ensure message order.*

Q: How to guarantee order in Kafka messages?
A:

- Message ordering is preserved **within a partition**.
- Use a consistent **key** when producing messages so they are routed to the same partition.

📦 Broker & Cluster

4. **Scenario:** *A broker crashes.*

Q: What happens to its partitions and data?
A:

- Kafka elects a new leader for each affected partition from in-sync replicas (ISRs).
- Consumers and producers are redirected to the new leaders.
- No data loss if replication is configured correctly.

5. **Scenario:** *You add a new broker. Data isn't balanced.*

Q: How do you balance partitions?
A:

- Use Kafka's `kafka-reassign-partitions.sh` tool to manually or automatically rebalance partitions across brokers.

---

6. **Scenario:** *Kafka is running out of disk space.*

Q: What actions do you take?
A:

- Check and adjust `log.retention.hours` or `log.retention.bytes`.
- Clean old topics or increase disk.
- Use compression (`compression.type=snappy/gzip`) to reduce disk usage.

📥 Consumer Side

7. Scenario: *Consumer restarts and reprocesses old data.*

Q: Why, and how do you fix it?
A:

- This can happen if auto-commit is disabled or offset was not committed.
- Set `enable.auto.commit=true` or manually commit offsets after processing.
- Ensure offsets are committed **after** successful processing to avoid duplicates.

---

8. Scenario: *Multiple consumers in a group, but some partitions are idle.*

Q: Why, and how to fix it?
A:

- Number of partitions < number of consumers.
- Each partition is assigned to one consumer only.
- Increase partition count or decrease consumer count for better utilization.

9. Scenario: *Consumer lag is increasing rapidly.*

Q: How do you troubleshoot?
A:

- Check consumer processing time—maybe it's slow.
- Increase number of partitions and consumers.
- Review thread pool usage and consumer group rebalances.

⚙️ Performance and Configuration

10. Scenario: *You want to reduce producer latency.*

Q: What Kafka configs do you tune?
A:

- Reduce `linger.ms` to flush messages sooner.
- Reduce `batch.size` to send smaller batches.
- Use `acks=1` or `acks=0` for lower latency (at the cost of reliability).
- Use async send for non-blocking I/O.

11. Scenario: *You want to maximize Kafka throughput.*

Q: What configuration changes help?
A:

- Increase `batch.size`, `linger.ms`, and use `compression.type=snappy`.
- Tune broker configs like `num.network.threads`, `socket.send.buffer.bytes`.
- Use multiple partitions to allow parallelism.

🔒 Reliability and Recovery

12. Scenario: *You want to avoid data loss in case of broker failure.*

Q: What configurations do you use?
A:

- Set **replication factor ≥ 2**.
- Set `min.insync.replicas` appropriately.
- Use `acks=all` and `enable.idempotence=true` on producer.

13. Scenario: *A topic has replication factor 3. Two brokers go down. Can you still consume data?*

A:

- Only if one broker (with leader replica) is alive and it's in the ISR.
- If the leader is not in ISR or down, the partition becomes unavailable.

🔐 Security Scenarios

14. Scenario: *Your producer gets "Not authorized to access topic".*

Q: What might be wrong?
A:

- The user doesn't have `Write` ACL on that topic.
- Check and update ACLs using Kafka's ACL CLI.
- Also check authentication via SASL or TLS.

📊 Monitoring and Ops

15. Scenario: *Consumer group is stuck and lag keeps increasing.*

Q: How do you debug this?

A:

- Check consumer logs for rebalance or deserialization errors.
- Restart the consumers.
- Use tools like Kafka Manager, Burrow, or Prometheus to monitor consumer lags.

🎯 Design Scenario

16. Scenario: *You're asked to design an exactly-once data pipeline with Kafka.*

Q: How do you do that?

A:

- Use Kafka transactional producers with `enable.idempotence=true`.
- Use `transactional.id` and send to output topic within a transaction.
- Consumers should use `read_committed` isolation level.
- Process and commit output within the same transaction.

1. **Explain Kafka transaction management with spring boot example?**

Kafka transaction management ensures **atomicity** in message processing, meaning either all operations succeed or none are applied. This is crucial for **exactly-once semantics (EOS)** in distributed systems.

How Kafka Transactions Work

1. **Transaction Initiation**: A producer starts a transaction using a unique `transactional.id`.
2. **Message Production**: Messages are sent within the transaction.
3. **Commit or Abort**: If all operations succeed, the transaction is committed; otherwise, it is aborted.

Spring Boot Kafka Transaction Example

Spring Boot simplifies Kafka transactions using `KafkaTransactionManager`. Here's how you can implement it:

Creating the Topic and replica details

```java
@SpringBootApplication
public class TransactionsService {

    public static void main(String[] args) {
        SpringApplication.run(TransactionsService.class, args);
    }

    @Bean
    public NewTopic transactionsTopic() {
        return TopicBuilder.name("transactions")
                .partitions(3)
                .replicas(1)
                .build();
    }

}
```

Configuring Kafka transactions in application.yml file

```yaml
spring:
  kafka:
    producer:
      key-serializer: org.apache.kafka.common.serialization.LongSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
      transaction-id-prefix: tx-
```

2. Define a Kafka Producer with Transactions

```java
@Service
public class KafkaProducerService {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public KafkaProducerService(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    @Transactional
    public void sendTransactionalMessage(String topic, String message) {
        kafkaTemplate.send(topic, message);
        // Other database operations can be included within the same transaction
    }
}
```

3. Handling Transactions in a Consumer

```java
@Service
public class TransactionsListener {

   private static final Logger LOG = LoggerFactory
           .getLogger(TransactionsListener.class);

   @KafkaListener(
           id = "transactions",
           topics = "transactions",
           containerGroup = "a",
           concurrency = "3")
   @Transactional
   public void listen(Order order) {
      LOG.info("{}", order);
   }
}
```

Key Benefits

- **Ensures atomicity** across multiple Kafka topics.
- **Prevents duplicate messages** using idempotent producers.
- **Supports distributed transactions** when combined with databases.

4. In kafka how to send the same data to multiple consumers?

🔄 Kafka Default Behavior

- If **multiple consumers are in the same consumer group**, Kafka will load-balance partitions among them → **each consumer gets part of the data**.
- If **consumers are in different consumer groups**, then each consumer will get the **full copy** of the data.

✅ Solution: Use **multiple consumer groups**

Each consumer (or group of consumers working together) should have its own **unique consumer group ID**.

java Copy Edit

```java
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "consumer-group-1");  // <-- use different group IDs
props.put("enable.auto.commit", "true");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("your-topic-name"));
```

Create a second consumer with `group.id = "consumer-group-2"` to receive the **same messages**.

Here's a complete **Java Spring Boot** example that demonstrates **sending the same Kafka message to multiple consumers** using **different consumer groups** — so **each consumer gets a full copy**.

✅ Overview

1. **ProducerController** – Sends messages to a Kafka topic.
2. **Two Consumer Services** – Each with a **different consumer group ID**, both receiving the same messages.

🧱 Maven Dependencies (`pom.xml`)

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
</dependencies>
```

⚙️ `application.yml`

```yaml
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer

topic:
  name: my-topic
```

🔥 Kafka Producer – `ProducerController.java`

java

```java
@RestController
@RequestMapping("/produce")
public class ProducerController {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Value("${topic.name}")
    private String topic;

    @PostMapping("/{message}")
    public ResponseEntity<String> sendMessage(@PathVariable String message) {
        kafkaTemplate.send(topic, message);
        return ResponseEntity.ok("Message sent to Kafka topic: " + message);
    }
}
```

🔥 Consumer 1 – `ConsumerGroupOne.java`

java

```java
@Service
public class ConsumerGroupOne {

    @KafkaListener(topics = "${topic.name}", groupId = "group-one")
    public void listen(String message) {
        System.out.println("Consumer Group One received: " + message);
    }
}
```

🔥 Consumer 2 – `ConsumerGroupTwo.java`

java

```java
@Service
public class ConsumerGroupTwo {

    @KafkaListener(topics = "${topic.name}", groupId = "group-two")
    public void listen(String message) {
        System.out.println("Consumer Group Two received: " + message);
    }
}
```

🏁 Start and Test

1. Make sure Kafka is running locally at `localhost:9092`.
2. Run the Spring Boot application.
3. Send a message using:

```bash
curl http://localhost:8080/produce/HelloKafka
```

4. You will see **both consumers** printing the same message.

✅ Output

```sql
Consumer Group One received: HelloKafka
Consumer Group Two received: HelloKafka
```

5. In Kafka how to identify and process the duplicate records?

Identifying and processing duplicate records in Kafka is a common concern when ensuring data integrity, especially with at-least-once delivery semantics. Kafka does not automatically deduplicate messages — it guarantees order and durability, but not uniqueness.

🔁 Why Do Duplicates Occur in Kafka?

Kafka guarantees **at-least-once** delivery by default. This means:

- Messages might be **delivered more than once** to the consumer.
- Duplicates may happen due to:
  - Consumer retry logic (e.g., exception in processing)
  - Network errors
  - Producer retries with same data (especially when retries are enabled)
  - Kafka rebalance

✅ Common Approaches to Detect and Handle Duplicates in Kafka

🔹 1. Use a Unique Identifier per Message

Each message should contain a **UUID or business key** (like order ID, transaction ID, user ID + timestamp, etc.).

Example Message Payload:

```json
{
  "id": "txn-12345",
  "amount": 100,
  "timestamp": "2025-07-14T12:30:00"
}
```

🔹 2. Maintain a Deduplication Store

You'll need to store **processed message IDs** somewhere and check incoming messages against it:

Option	Pros	Cons
In-memory Set	Fast, simple	Loses data on restart, limited size
Redis	Fast, persistent	Extra infra
Database (MySQL)	Durable, queryable	Slower, may need indexing
Kafka Compact Topic	Native, scalable	More complex logic needed

🔹 3. Spring Kafka Example – Deduplication with Redis

💡 **Use case: Don't process the same ID twice**

🧱 1. Setup `pom.xml` dependencies

```xml
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

⚙️ 2. application.yml

yaml

```yaml
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: dedup-group
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

  redis:
    host: localhost
    port: 6379
```

📥 3. Kafka Consumer with Redis Deduplication

java

Copy

```java
@Service
public class OrderConsumer {

    @Autowired
    private StringRedisTemplate redisTemplate;

    private static final String DEDUP_KEY_PREFIX = "processed:";

    @KafkaListener(topics = "orders", groupId = "dedup-group")
    public void consume(String jsonMessage) {
        OrderEvent order = parse(jsonMessage);
        String redisKey = DEDUP_KEY_PREFIX + order.getEventId();

        // Check and insert if not already present
        Boolean isFirstSeen = redisTemplate.opsForValue()
                .setIfAbsent(redisKey, "1", Duration.ofHours(1)); // expires in 1 hour

        if (Boolean.TRUE.equals(isFirstSeen)) {
            // ✅ Process the message
            System.out.println("Processing new order: " + order.getEventId());
            // ... your business logic here ...
        } else {
            // ❌ Duplicate - skip
            System.out.println("Duplicate order skipped: " + order.getEventId());
        }
    }

    private OrderEvent parse(String json) {
        try {
            return new ObjectMapper().readValue(json, OrderEvent.class);
        } catch (Exception e) {
            throw new RuntimeException("Invalid JSON: " + json);
        }
    }
}
```

```
    private OrderEvent parse(String json) {
        try {
            return new ObjectMapper().readValue(json, OrderEvent.class);
        } catch (Exception e) {
            throw new RuntimeException("Invalid JSON: " + json);
        }
    }
}
```

4. OrderEvent.java (POJO)

```java
public class OrderEvent {
    private String eventId;
    private String userId;
    private double amount;
    private String timestamp;

    // Getters & setters
}
```

🔹 4. Kafka Exactly-Once Semantics (EOS) *(Advanced)*

If you're using Kafka >= 2.5, you can enable **exactly-once** using:

- Idempotent producers
- Transactional processing (Kafka + DB or Kafka-to-Kafka)
- Spring Kafka transaction support

Limitation: EOS guarantees no duplication **only inside Kafka pipelines**, not external systems unless managed carefully.

✅ Summary

Strategy	When to Use
Unique ID + Redis	Fast, simple deduplication
Unique ID + DB	Durable deduplication, fits business data model
Kafka Compacted Topic	Kafka-native deduplication
Exactly Once Semantics (EOS)	Full guarantee of no duplicates (with cost)

6. Explain Consumer group?

Kafka Consumer Groups is **essential for scaling, reliability, and performance tuning** in Kafka-based systems.

Let's break down **Kafka Consumer Groups** from the ground up, with **real-world examples, diagrams, and key behaviors**.

🧠 What is a Consumer Group?

A **Consumer Group** in Kafka is a **logical group of consumers** that **work together to consume data from a Kafka topic.**

- **Each consumer in the group** reads data from a **subset of the partitions.**
- **Kafka ensures** that **each partition is read by only one consumer in the group** (at a time).

> 💡 **Key Rule:** A **partition** can be consumed by **only one consumer** in a group at any time, but **can be read by multiple groups** independently.

🧱 Components in the Model

- **Kafka Topic:** Divided into multiple **partitions.**
- **Consumer:** Application instance that reads from a partition.
- **Consumer Group:** Identified by `group.id` config; a group of consumers working together.

📦 Example Setup

bash Copy Edit

```
Topic: "orders"
Partitions: 3 (P0, P1, P2)

Consumer Group ID: "group-A"
Consumers: C1, C2
```

Partition	Assigned to
P0	C1
P1	C2
P2	C1 (or C2)

🔄 How Kafka Assigns Partitions to Consumers

Kafka uses a **partition assignment strategy** (like **range** or **round robin**) to distribute topic partitions across consumers in a group.

Partition Rebalancing Triggers:

- New consumer joins the group.
- A consumer leaves or crashes.
- Topic partitions change.
- Admin triggers rebalancing (e.g., config change).

🧬 Behavior Based on Grouping

✅ **Same Consumer Group** (Shared Workload)

All consumers in the **same group** coordinate and **split the workload.**

```bash
group.id = "group-orders"
```

- Kafka ensures each partition is read by only one consumer.
- Best for **scaling consumers horizontally.**
- Enables **load balancing.**

✅ **Different Consumer Groups** (Full Copy to Each)

Each consumer group **acts independently** and **gets a full copy** of all data.

```bash
group.id = "group-A"  (App1)
group.id = "group-B"  (App2)
```

- All groups will **consume all records.**
- Best for **fan-out architecture** (e.g., billing, audit, analytics all consume same event stream).

🔁 At-Least-Once Processing Guarantee

Each consumer group maintains **offsets** in Kafka to know **where it left off**.

- Stored in Kafka topic: `__consumer_offsets`
- Can be **committed automatically** (`enable.auto.commit=true`) or manually.

⚠️ Misconfigurations to Avoid

Issue	Cause	Fix
Duplicate processing	Same group ID across independent services	Use **separate group IDs**
Missing messages	Too few consumers for number of partitions	Add more consumers or scale partitions
Lag	Slow processing or offset not committed	Tune processing logic or commit offsets properly
Partition stuck	Only one consumer but multiple partitions	Use multiple consumers or multi-threaded consumer logic

🔄 Real-Life Scenarios

Scenario 1: Scaling Workers

- App processes millions of orders per day.
- Use **5 partitions** and **5 consumers in same group**.
- Each consumer processes 1 partition in parallel.

Scenario 2: Multiple Applications Need Same Events

- Billing service and analytics service both want order data.
- Use **two consumer groups**: `"billing-group"` and `"analytics-group"`.
- Each app gets a **full copy**.

🔎 Monitoring Consumer Groups

Kafka provides tools to inspect:

```bash
kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --describe \
  --group group-orders
```

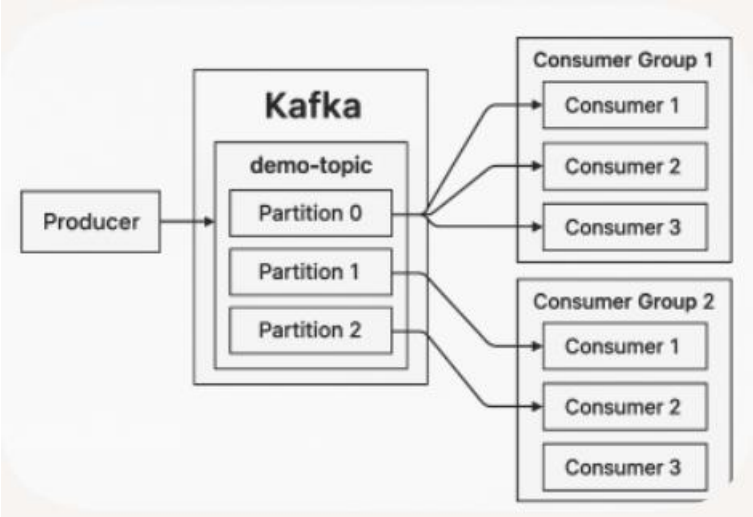
This shows:

- Current offset
- Log end offset
- Lag (difference)
- Partition assignment

✅ Summary

Concept	Explanation
Consumer Group	A set of consumers collaborating to read from a topic
Group ID	Unique name that defines a consumer group
Partition Assignment	Kafka assigns partitions to consumers in a group
Scaling	Add more consumers to same group to parallelize workload
Fan-out	Use different group IDs to send same message to multiple consumers
Offset Management	Kafka stores offset per group per partition in `__consumer_offsets`

8. How do consumer groups manage message consumption?



Consumer groups in Kafka are like coordinated teams that divide and conquer message consumption from a topic. Here's how they manage it efficiently

🧠 Partition Assignment Strategy

- Kafka assigns **partitions** of a topic to consumers within a group.
- Each partition is consumed by **only one consumer** in the group at a time.
- If there are more consumers than partitions, some consumers remain **idle** [1].
- If a consumer fails or leaves, Kafka **rebalances** the group and reassigns partitions.

🧭 Offset Tracking

- Kafka tracks the **offset** (position) of each consumer in the internal `__consumer_offsets` topic. [2]
- Consumers can **auto-commit** offsets or **manually commit** them after processing messages.
- This ensures **fault tolerance**—if a consumer crashes, another can resume from the last committed offset. [2]

Rebalancing Mechanism

- Triggered when a consumer **joins, leaves, or crashes**.
- Kafka pauses consumption, reassigns partitions, and resumes processing.
- The **group coordinator** and **group leader** handle this behind the scenes.

Configuration Highlights

Property	Purpose
group.id	Identifies the consumer group
enable.auto.commit	Enables automatic offset commits
auto.offset.reset	Defines behavior when no offset is found
session.timeout.ms	Time to detect consumer failure
heartbeat.interval.ms	Frequency of heartbeat to group coordinator

8. How do producers interact with consumer groups?

In Kafka, **producers and consumer groups interact indirectly through topics and partitions**, not through direct communication. Here's how it works:

Kafka Interaction Model

- **Producers** publish messages to a **topic**.
- **Topics** are divided into **partitions**.
- **Consumer groups** subscribe to topics and consume messages from partitions.

Key Concepts

Component	Role
Producer	Sends messages to a topic.
Topic	Logical channel for messages, split into partitions.
Partition	Unit of parallelism; each message lands in one partition.
Consumer Group	Set of consumers that share the work of reading from a topic.
Offset	Position of a message in a partition; used to track consumption.

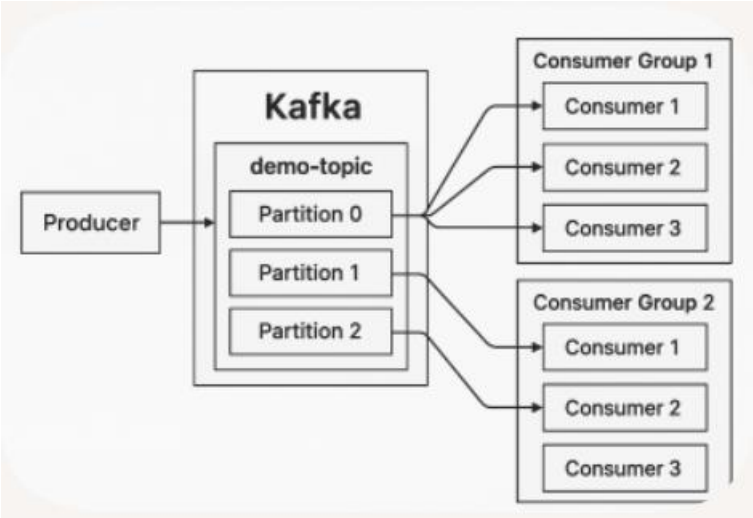
📬 How Messages Flow

1. **Producer sends a message** to a topic (optionally with a key to control partitioning).
2. Kafka **stores the message** in one of the topic's partitions.
3. **Consumers in a group** are assigned partitions (one partition per consumer).
4. Each consumer **pulls messages** from its assigned partition.
5. Kafka ensures **each message is delivered to only one consumer** in the group.

🧠 Important Notes

- **Producers don't know or care about consumer groups**. They just send data to topics.
- **Multiple consumer groups** can read the same data independently (like a broadcast).
- **Within a group**, consumers divide the partitions among themselves for scalability
- Kafka uses **offsets** to track what each consumer has read, stored in `__consumer_offsets`.

10. Explain multi consumer group consumption approach?



📊 Diagram Overview Recap

- A single **producer** sends data to `demo-topic`, which is split into **three partitions**.
- Two distinct **consumer groups** (Group 1 and Group 2), each with **three consumers**, independently consume from those partitions.
- Each consumer within a group is mapped to one partition—this ensures parallelism within the group.

🏭 Real-Time Example: Smart Factory Monitoring

Imagine a **smart manufacturing plant** with sensors installed on various machines. Each sensor sends its **real-time temperature readings**, vibration metrics, and operational status to Kafka.

🔸 The Producer

A microservice or edge device collects and pushes data every second to the Kafka topic `demo-topic`.

🔹 Kafka Topic & Partitions

`demo-topic` splits the incoming stream into 3 partitions to distribute load:

Partition	Type of Machine Data
0	Temperature readings
1	Vibration metrics
2	Operational status signals

👥 Consumer Group 1 – **Live Dashboard System**

- Displays real-time metrics on screens inside the control room.
- Consumers fetch partition-specific data:
  - Consumer 1 handles temperatures.
  - Consumer 2 tracks vibrations.
  - Consumer 3 watches machine statuses.

👥 Consumer Group 2 – **Alert & Prediction System**

- Evaluates data to trigger alerts or predict equipment failures.
- Consumers mirror Group 1's partition handling, but independently:
  - Consumer 1 analyzes temperature spikes.
  - Consumer 2 looks for unusual vibrations.
  - Consumer 3 detects abnormal status codes.

🔁 Since both groups are independent:

- **Group 1 ensures visibility** and monitoring.
- **Group 2 ensures action**, like predictive maintenance or emergency response.

⚡ Takeaway

This setup lets different systems consume the **same data stream without interference**, all while scaling horizontally. It's like two teams reading the same set of logs—but for entirely different purposes. 💡

11. What happens when a consumer joins or leaves during processing?

When a Kafka consumer **joins or leaves a group during processing**, the system initiates a process called **rebalancing**, which temporarily disrupts message consumption to redistribute partitions among active consumers. Here's how it unfolds:

🔄 What Triggers Rebalancing?

- **Consumer joins** the group (e.g., scaling up).
- **Consumer leaves** due to crash, shutdown, or timeout.
- **New partitions** are added to the topic.
- **Heartbeat failures** or long poll intervals cause Kafka to evict idle consumers.

⚙️ What Happens During Rebalancing?

Step	Description
1. **Pause Consumption**	All consumers stop reading messages temporarily.
2. **Partition Reassignment**	Kafka redistributes topic partitions among the current consumers.
3. **Resume Consumption**	Consumers start processing messages from their new assignments.

This is often called a **"stop-the-world" event**, especially with eager rebalancing, where all consumers give up their partitions at once. [1] [2]

🧠 Rebalance Strategies

Kafka supports several assignment strategies:

- **RangeAssignor**: Assigns contiguous partitions.
- **RoundRobinAssignor**: Distributes partitions evenly.
- **StickyAssignor**: Minimizes partition movement.
- **CooperativeStickyAssignor**: Allows **incremental rebalancing**, so only affected consumers pause. [1]

🧪 Real-Time Impact Example

Imagine a **real-time fraud detection system**:

- Consumers analyze transactions from different regions.
- If one consumer crashes, Kafka reassigns its region's partition to another.
- During rebalancing, detection may pause briefly—potentially delaying alerts.

🛡️ How to Minimize Disruption

- Use **cooperative rebalancing** to avoid full pauses.
- Configure `max.poll.interval.ms` and `session.timeout.ms` wisely.
- Consider **static group membership** with `group.instance.id` to retain partition ownership during restarts. [1]